

# **NetLogger Cookbook**

<b>COLLABORATORS</b>		
	<i>TITLE :</i> NetLogger Cookbook	<i>REFERENCE :</i>
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY	Dan Gunter	May 29, 2009

<b>REVISION HISTORY</b>			
NUMBER	DATE	DESCRIPTION	NAME

NUMBER	DATE	DESCRIPTION	NAME

---

## Contents

<b>1 Preface</b>	<b>5</b>
1.1 Conventions . . . . .	5
<b>2 NetLogger Pipeline</b>	<b>5</b>
2.1 Prerequisites . . . . .	5
2.2 Getting the code . . . . .	5
2.3 Pipeline parser (nl_parser) . . . . .	5
2.3.1 Running in standalone mode . . . . .	6
2.3.2 Get help on a parser module . . . . .	6
2.3.3 Passing parameters to a parser . . . . .	6
2.3.4 Running from a configuration file . . . . .	7
2.3.5 Writing parser modules . . . . .	9
2.3.5.1 Parser module unit tests . . . . .	10
2.4 Pipeline Database Loader (nl_loader) . . . . .	11
2.4.1 Example 1: SQLite . . . . .	12
2.4.2 Example 2: MySQL . . . . .	12
2.5 Pipeline management script (nl_pipeline) . . . . .	13
2.5.1 Configuring the pipeline . . . . .	13
2.5.2 Running the pipeline . . . . .	13
2.5.3 Manually loading data into the pipeline . . . . .	14
2.6 Miscellaneous tasks . . . . .	15
2.6.1 Dumping and reloading a MySQL database . . . . .	15
2.6.2 Using Syslog-ng . . . . .	15
<b>3 NetLogger Analysis</b>	<b>15</b>
3.1 Lifelines . . . . .	15
3.2 Database queries . . . . .	16
3.2.1 Database event types . . . . .	16
<b>4 Logging API examples</b>	<b>17</b>
4.1 C API . . . . .	17
4.1.1 Basic usage . . . . .	17
4.1.2 Format a string . . . . .	19
4.1.3 Transfer API . . . . .	19
4.2 Java API . . . . .	23
4.2.1 Basic usage . . . . .	23
4.2.2 Log4J . . . . .	24
4.3 Perl API . . . . .	24

4.3.1	Basic usage . . . . .	24
4.4	Python API . . . . .	25
4.4.1	Basic usage . . . . .	25
4.4.2	Format strings . . . . .	25

## 1 Preface

This set of examples, or cookbook, show how to use parts of the NetLogger Toolkit. For more details, downloads, etc. see the NetLogger home page at <http://acs.lbl.gov/NetLoggerWiki/>.

### 1.1 Conventions

**Italic** Used for file and directory names, email addresses, and new terms where they are defined.

**Constant Width** Used for code listings and for keywords, variables, functions, command options, parameters, class names, and HTML tags where they appear in the text. Used with double quotes for literal values like “True”, “10” and “netlogger.modules”. In code listings, user input to the terminal will be prefixed with a “\$”.

**Constant Width Italic** Used to indicate items that should be replaced by actual values.

**Link text** Used for URLs and cross-references.

## 2 NetLogger Pipeline

### 2.1 Prerequisites

**Python 2.5** Many tools are written in Python, and some Python 2.5+ features are used

**UNIX** The interaction between the nl\_pipeline and nl\_parser/nl\_loader tools uses UNIX signals. More generally, although these tools should in theory mostly work in Windows, all development and testing has been done on Linux and Mac OSX.

### 2.2 Getting the code

This section shows how to get the latest NetLogger code from subversion and then set up your environment for running tools from the checked-out subversion workspace. This step should be done (once) before running any of the other examples in this section.

1. Check out the code from subversion

```
svn co https://cvs.globus.org/repos/netlogger/trunk netlogger-trunk
```

2. Set up environment

```
cd netlogger-trunk/python/  
source dev-setup.sh
```

### 2.3 Pipeline parser (nl\_parser)

The NetLogger Python tools include parsing framework. The framework has two principal goals: make it easy to flexibly apply a variety of parsers to a set of log files and make it easy to plug in your own parsers. The front-end program to do all this is called *nl\_parser*.

The *nl\_parser* can run in one of two modes:

**standalone mode** Read a file or files and parse with a single parser type, also called a *parser module*. All options are given on the command line.

**pipeline mode** Match parsers against files flexibly, optionally tailing input files and rolling over output files. Use a configuration file for all options.

### 2.3.1 Running in standalone mode

Once you have [set up your development environment](#), running the parser in standalone mode is mostly a matter of picking the name of your parser module and feeding it the file.

1. Create a "Best Practices" file

```
$ nl_write -n 10 > /tmp/sample_file.bp
```

2. Parse it with the `bp` module (a no-op, since the input and output are both the same format; but the idea is the same no matter the input format)

```
$ nl_parser -m bp /tmp/sample_file.bp
```

### 2.3.2 Get help on a parser module

The list of available built-in modules is currently available only by listing the contents of the `netlogger/parsers/modules` directory. Once you know the name of the module, you can get help on its parameters (if any) by combining the `-i`/`--info` flag with the `-m`/`--module` option whose value is the module you want to describe:

```
$ nl_parser -m bp -i
Module      : netlogger.parsers.modules.bp
Description: Parse Best-Practices logs into Best-Practices logs.
Parameters :
  - has_gid {True,*False*}: If true, the "gid=" keyword in the input will be
    replaced by the currently correct "guid=".
  - verify {*True*,False}: Verify the format of the input, otherwise simply
    pass it through without looking at it.
```

- Notice that the default option value is surrounded with \*asterisks\*.
- For boolean parameters, any of 0/1, no/yes, false/true are equivalent.

### 2.3.3 Passing parameters to a parser

The parsers understand the parameters listed in their help message. To show how to use these parameters, let's add a non-verifying line to the log we created above and see if turning off the `verify` option gets the parser to ignore it.

1. Add a "bad" Best Practices log entry:

```
$ echo "Hello, world" >> /tmp/sample_file.bp
```

2. Try with default verification settings. The output should not include the "Hello, world" log entry.

```
$ nl_parser -m bp /tmp/sample_file.bp
```

3. Now, turn off verification.

```
$ nl_parser -m bp -p verify=no /tmp/sample_file.bp
```

You should see the "Hello, world" line at the end.

Any parameter not recognized by the parser (and also not specially known to the base parser) will be simply added to each output log message. This can be a handy way to get run-time information into the logs. For example, to add the host name to our output from above:

```
$ nl_parser -mbp /tmp/sample_file.bp -p host='hostname'
```

### 2.3.4 Running from a configuration file

When running from a configuration file, all the options go in the configuration file itself, making the invocation of the nl\_parser simple.

```
$ nl_parser -c $config_file
```

The nl\_parser configuration file uses the INI syntax variant parsed by the [ConfigObj](#) third-party Python library. It has one [*global*] section and then one section per parser type. Each type of parser can be assigned to one or more files. One type of parser, the *dynamic* parser, is really a meta-parser that, for each of its input files, chooses the actual parser module on a per-line basis.

The main use of the configuration file is for the *pipeline mode* of the nl\_parser. In this mode, the nl\_parser is not run directly but instead is forked as a daemon by nl\_pipeline. But except for a couple of attributes in the [*global*] section, the configuration for running the nl\_parser directly is the same. For simplicity, that is what we will show here.

1. Start with the following log files in */tmp/logs*.

- *gridftp.log*: local GridFTP log

```
DATE=20070215110703.2102 HOST=pdsfgrid1.nersc.gov PROG=globus-gridftp-server NL.EVNT= ←
FTP_INFO START=20070215110702.763298 USER=jschmoe FILE=/tmp/afile BUFFER=0 BLOCK ←
=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST=[129.79.4.64] TYPE=RETR CODE ←
=226
DATE=20070215110704.2205 HOST=pdsfgrid1.nersc.gov PROG=globus-gridftp-server NL.EVNT= ←
FTP_INFO START=20070215110702.763298 USER=jschmoe FILE=/tmp/afile BUFFER=0 BLOCK ←
=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST=[129.79.4.64] TYPE=RETR CODE ←
=226
DATE=20070215110705.9731 HOST=pdsfgrid1.nersc.gov PROG=globus-gridftp-server NL.EVNT= ←
FTP_INFO START=20070215110702.763298 USER=jschmoe FILE=/tmp/afile BUFFER=0 BLOCK ←
=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST=[129.79.4.64] TYPE=RETR CODE ←
=226
```

- *remote.host.log*: Logs from a remote host, received through syslog-ng, that have both GridFTP and BeStMan log entries. The header for each line has either "gridftp\_log" or "bestman\_log" in it.

```
2008-07-23 INFO pbs_log:07/23/2008 00:43:53;S;545489.myhost.domain.com;user=foo group ←
=bar account=testrepo jobname=STDIN queue=debug ctime=1216799025 qtime=1216799025 ←
etime=1216799025 start=1216799033 owner=foo@myhost.domain.com exec_host= ←
nodename343/1+nodename343/0 Resource_List.neednodes=nodename343:ppn=2 ←
Resource_List.nodect=1 Resource_List.nodes=1:ppn=2 Resource_List.walltime=00:30:00 ←
.

2008-08-22 INFO gridftp_log:DATE=20080822110703.2102 HOST=pdsfgrid1.nersc.gov PROG= ←
globus-gridftp-server NL.EVNT=FTP_INFO START=20080822110702.763298 USER=jschmoe ←
FILE=/tmp/afile BUFFER=0 BLOCK=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST ←
=[129.79.4.64] TYPE=RETR CODE=226
2008-08-22 INFO unknown_log>Hello, world!
2008-08-22 INFO gridftp_log:DATE=20080822110704.2205 HOST=pdsfgrid1.nersc.gov PROG= ←
globus-gridftp-server NL.EVNT=FTP_INFO START=20080822110702.763298 USER=jschmoe ←
FILE=/tmp/afile BUFFER=0 BLOCK=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST ←
=[129.79.4.64] TYPE=RETR CODE=226
2008-08-22 INFO gridftp_log:DATE=20080822110705.9731 HOST=pdsfgrid1.nersc.gov PROG= ←
globus-gridftp-server NL.EVNT=FTP_INFO START=20080822110702.763298 USER=jschmoe ←
FILE=/tmp/afile BUFFER=0 BLOCK=262144 NBYTES=55 VOLUME=/ STREAMS=1 STRIPES=1 DEST ←
=[129.79.4.64] TYPE=RETR CODE=226
```

2. Then create a configuration file in */tmp/logs/test.conf*. See annotations below for a description.

```
[global]
files_root = /tmp/logs # Prepend this directory to input file names
# Would add these in pipeline mode:
# eof_event = True
```

```

# tail = True
output_file = /tmp/logs/parsed.bp # Output file

[parser_1]
[[gridftp]]
files=gridftp.log # Pattern to match for gridftp

[parser_2]
files = *.host.log # Pattern to match for syslog files
# Header for each line. Three "named groups" will be extracted:-
# ts, level, and app. These can be used as keywords in the subsequent
# $$[[[match]]]$$ section.
pattern = "(?P<ts>\S+) (?P<level>[A-Z]+) (?P<app>\S+):"
[[gridftp]]
[[[match]]]
# Regular expression to match the named group
# against. If multiple keyword=value pairs are present
# in this section, then ALL of them must match.
app = "gridftp_log"
[[pbs]]
[[[match]]]
app = "pbs_log"
[[generic]] # Put log contents in "msg" field
[[[match]]]
# Fall-through: matches anything

# A basic logging configuration that puts all the messages in.
# a file.

[logging]
[[loggers]]
[[[netlogger]]]
level=DEBUG
handlers=h1
qualname=netlogger
[[handlers]]
[[[h1]]]
class="FileHandler"
level=DEBUG
args="('/tmp/logs/netlogger-all.bp', 'a')"

```

- Run the `nl_parser` with this configuration. This will put its output in `/tmp/logs/parsed.bp`.

```
$ nl_parser -c /tmp/logs/test.conf
```

The output is a little messy, but you can look at the output with the `nl_view` tool. For example, to see the default timestamp and event fields along with host and user:

```
$ nl_view -a user -a host -a user -w12 parsed.bp
2008-07-23 07:43:53.000000Z pbs.job.S      foo
2008-08-22 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
2008-09-23 18:47:10.388647Z event
2008-08-22 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
2008-08-22 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
2007-02-15 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
2007-02-15 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
2007-02-15 11:07:02.763298Z FTP_INFO      jschmoe pdsfgrid1.nersc.gov
```

### 2.3.5 Writing parser modules

Parser modules are just Python modules (.py files) that follow a simple set of rules, illustrated with the example parser module here.

Here is an overview of what your parser plug-in should look like. See the annotations below for details.

1. The main class must be called `Parser`, and must inherit from `netlogger.parsers.base.BaseParser`
2. The `Parser` constructor should take a positional `fileobj`, then 0 or more named parameters, and then keyword arguments to pass up to the `BaseParser`.
3. The parsing logic should go in a `Parser` method called `process`, which takes one argument that is the next line of input
4. The `process` method must return a sequence of 0 or more dictionaries, each of which must have the keyword "ts" and the keyword "event".

The module is going to be called `ps_parser.py` because it is designed to process the output of the command:

```
$ ps -o pid,comm
```

---

#### Example 2.1 Parser module ps\_parser.py

---

```
"""Parse result of 'ps -o pid,comm'"""
import time
# Import the superclass
from netlogger.parsers.base import BaseParser
# Required class 'Parser', which inherits from 'BaseParser'
class Parser(BaseParser):
    # Your parser must take a positional fileobj, then 0 or more
    # named parameters, and then keyword arguments to pass up
    # to the 'BaseParser'
    def __init__(self, f, my_param = 'foo', **kw):
        # Pass the fileobj and keywords up to the BaseParser. The fullname is
        # used as the qualname argument to the Python logging.getLogger()
        # BaseParser.__init__(self, f, fullname=__name__, **kw)
    # This method does all the work of parsing.
    # Each invocation gets a single line from the input sources, and is
    # expected to return a sequence of 0 or more dictionaries that will be
    # turned into NetLogger log entries.
    def process(self, line):
        # If there is nothing to return, just return an empty sequence
        if line.startswith("PID"): return ()
        # Parsing logic..
        pid, comm = line.split(' ',1)
        # Build the return value. The value for the 'ts' keyword should be
        # either the floating-point number of seconds since the UNIX epoch
        # (midnight on Jan 1, 1970) or a string formatted as an ISO 8601
        # timestamp like '2008-06-05T19:28:26.718230Z'.
        # The timezone ('Z') can optionally be replaced with a +/-HH:MM.
        # The value for the 'event' keyword should be a unique name using
        # the Java class name convention of prefixing with a reversed
        # DNS ('org.mydomain', in this case). If the event is part of a pair
        # indicating the start and end of something, the pair of events should
        # use the suffixes '.start' and '.end', respectively.
        # See the CEDPS Logging Best Practices guide for details on the above.
        d = dict(ts=time.time(), event="org.mydomain.ps.item",
                 pid=int(pid), command=comm)
        # Return a sequence. Remember to always return a sequence even if
        # you have only one return value.
        return (d,)
```

- Go to the directory with `ps_parser.py`
- Run the code. Since you are not using one of the parser distributed with NetLogger, you need to add the `-x—external` flag to the `nl_parser` invocation. Otherwise, using your parser is just the same as using a built-in parser.

```
$ ps -o pid,comm | nl_parser -x -m ps_parser
```

- The output should look something like the following.

```
ts=2008-06-05T20:53:06.301956Z event=org.mydomain.ps.item level=Info pid=28418 command=- ↵
    bash
ts=2008-06-05T20:53:06.302205Z event=org.mydomain.ps.item level=Info pid=28540 command=/ ↵
    System/Library/.../MacOS/Python
ts=2008-06-05T20:53:06.302279Z event=org.mydomain.ps.item level=Info pid=28649 command=- ↵
    bash
ts=2008-06-05T20:53:06.302343Z event=org.mydomain.ps.item level=Info pid=28882 command=/ ↵
    System/Library/.../MacOS/Python
ts=2008-06-05T20:53:06.302407Z event=org.mydomain.ps.item level=Info pid=29176 command=- ↵
    bash
ts=2008-06-05T20:53:06.302469Z event=org.mydomain.ps.item level=Info pid=29583 command=- ↵
    bash
ts=2008-06-05T20:53:06.302531Z event=org.mydomain.ps.item level=Info pid=33019 command=/ ↵
    System/Library/.../MacOS/Python
ts=2008-06-05T20:53:06.302595Z event=org.mydomain.ps.item level=Info pid=31792 command=- ↵
    bash
ts=2008-06-05T20:53:06.302656Z event=org.mydomain.ps.item level=Info pid=31878 command=ssh
ts=2008-06-05T20:53:06.303252Z event=org.mydomain.ps.item level=Info pid=32482 command=- ↵
    bash
ts=2008-06-05T20:53:06.303470Z event=org.mydomain.ps.item level=Info pid=32568 command=/ ↵
    System/Library/.../MacOS/Python
```

### 2.3.5.1 Parser module unit tests

NetLogger has a unit test base class in the module “`netlogger.tests.testBase`” that can make writing tests for your own parsers much easier. The procedure for writing a unit test that uses this base class is: . Make a data file with an example log. Put this file in a subdirectory called `data`, i.e., if your module is in `DIR` then the example log should be placed in `DIR/data/`. . Write a Python module using the `unittest` framework. Its main class should inherit from `netlogger.tests.testBase.BaseParserTestCase`. . Write your test functions as methods of this class, prefixed with `test`, e.g. a test of `foo` should be in a method called “`testFoo()`”. Run the test case.

Here is an example test case for [the example parser](#).

---

**Example 2.2** Test case for ps\_parser.py

---

```
#!/usr/bin/env python
"""
Unitests for ps_parser.py
"""

import unittest
from netlogger.tests import testBase
import ps_parser

class TestCase(testBase.BaseParserTestCase):
    """Unit test cases.

    """
    basename = "ps_parser-"
    parser_class = ps_parser.Parser

    def testBasic(self):
        """Basic test of ps_parser

        """
        # the full path should be data/ps_parser-sample.log
        filename = "sample.log"
        # count lines in file
        expected = len(list(file(self.getFullPath(filename))))
        # subtract one for ps's header line
        expected = expected - 1
        # Check if all lines got parsed
        self.checkGood(filename=filename, num_expected=expected)

    # Boilerplate to run the tests
def suite():
    return testBase.suite(TestCase)
if __name__ == '__main__':
    testBase.main()
```

---

To run the example, do the following.

1. Make a temporary directory and change to it
2. Get the test code

```
$ wget http://acs.lbl.gov/NetLoggerWiki/images/d/d9/TestPsParser.py
$ mv TestPsParser.py testPsParser.py
```

3. Get the parser code

```
$ wget http://acs.lbl.gov/NetLoggerWiki/images/6/66/Ps_parser.py
$ mv Ps_parser.py ps_parser.py
```

4. Create the data file

```
$ ps -o pid,comm > data/ps_parser-sample.log
```

5. Run the parser

```
$ python testPsParser.py
```

---

## 2.4 Pipeline Database Loader (nl\_loader)

The nl\_loader inserts NetLogger Best Practices records into a database.

---

### 2.4.1 Example 1: SQLite

This section shows how to load some data into the NetLogger database, starting from scratch, i.e. without having NetLogger installed. It uses a zero-configuration database, [the embedded database SQLite](#). In order to run the queries in steps 5 and 6, you will need to download and install SQLite for your system.

1. Check out NetLogger from the Subversion repository: see [Getting the code](#).
2. Get a small (~300 record) parsed log file (this is the result of running a non-BP log file through one of our parsers)

```
wget http://acs.lbl.gov/~ksb/osg-bp.log.86
```

3. Load that file into a new sqlite db (-C creates the tables)

```
nl_loader -u sqlite://./osg.sqlite -i osg-bp.log.86 -C
```

4. You could now run sqlite3 commands:

```
sqlite3 ./osg.sqlite "select count(*) from event"
```

OR use nl\_dbquery, which expects a .conf file

```
cp netlogger-trunk/python/examples/config/nl_dbquery_pegasus.conf nl_dbquery.conf
```

5. That .conf file then needs to be edited for queries specific to that data. Adding in the above query as the first one in the file, nl\_dbquery is run like this:

```
nl_dbquery -u sqlite://./osg.sqlite -q 1
```

### 2.4.2 Example 2: MySQL

This section shows how to create a *fake* database to test some queries and dump/load, using a MySQL back-end. Refer to the example above for quick-start instructions to download NetLogger itself.

- MySQL: put username and password in ~/.my.cnf

```
$ cat ~/.my.cnf
[client]
user = cedps
host = localhost
password = "foo"
```

- Create data

```
$ nl_write -in 100 > fake.log
```

- Set up database permissions

```
$ mysql -u root -p mysql
mysql> drop database fake;
mysql> create database fake;
mysql> grant all on fake.* to 'cedps'@'localhost';
```

- Load the data into the database (-C creates the tables)

```
$ nl_loader -u mysql://localhost -p db=fake -C -i fake.log
```

- Try a query

```
$ mysql -D fake -e "select count(*) from attr;"
```

count(*)
200

## 2.5 Pipeline management script (nl\_pipeline)

This section describes how to manage the NetLogger pipeline (nl\_pipeline). Throughout this section we will note the root directory for your NetLogger configuration, logs as \$DIR. This way, you can use the examples verbatim by first setting DIR to the desired document root, as shown in the following.

```
# for UNIX sh/bash/ksh, etc.
sh$ DIR=/opt/netlogger ; export DIR
# or, if you use a C-Shell derivative
csh% setenv DIR /opt/netlogger
```

### 2.5.1 Configuring the pipeline

The nl\_pipeline forks off the nl\_parser and nl\_loader and runs them as daemons. For simplicity, it assumes that the configuration for both of these tools is in the same directory and named *nl\_parser.conf* and *nl\_loader.conf* respectively.

The base directory for the configuration files is given as the *-c*/*--config* option to the nl\_pipeline command. If this directory is \$DIR/etc, the default directory layout for the remaining files is as follows.

```
$DIR
|
+- etc
|   |
|   +- nl_parser.conf
|   +- nl_loader.conf
|
+- var
    |
    +- log
        |
        +- nl_parser.log
        +- nl_loader.log
        +- nl_pipeline.log
    |
    +- run
        |
        +- nl_parser.pid
        +- nl_loader.pid
        +- nl_pipeline.pid
```

### 2.5.2 Running the pipeline

First, create configuration files as described in [Configuring the pipeline](#). This will place everything in a subdirectory of \$DIR. Run the pipeline with the "-n" option to see what would be done.

```
nl_pipeline -n -v -c $DIR/etc -l $DIR/var/log -p $DIR/var/run -i 30
```

This will test the existence of the relevant directories, etc., and output something like this:

```
would run:
/usr/bin/nl_parser -d $DIR/var/run/nl_parser.pid -c $DIR/etc/nl_parser.conf
/usr/bin/nl_loader -d $DIR/var/run/nl_loader.pid -c $DIR/etc/nl_loader.conf
# reload parser configuration every 30 seconds
```

Next try running each of those command-lines individually, but without the -d flag so any errors just get dumped to stderr. Run them in the background so you can send them a signal (next step).

```
/usr/bin/nl_parser -c $DIR/etc/nl_parser.conf &
/usr/bin/nl_loader -c $DIR/etc/nl_loader.conf &
# If these both run without error, try sending each of them a SIGHUP.
# This signal causes them to re-read their configuration files.
kill -HUP %1
kill -HUP %2
```

Try sending the nl\_parser a SIGUSR1, which tells it to rotate its output file, e.g. with `kill -USR1 %1`. If things fail you should get errors on stderr as well as some information in the logs at `$DIR/var/log/nl_parser.log` and `nl_loader.log`. Once this all seems to run OK, kill the loader and parser (for real), erase the .state files, and start the pipeline:

```
kill %1 %2
rm -f $DIR/var/*.state
nl_pipeline -v -c $DIR/etc -l $DIR/var/log -p $DIR/var/run -i 30
```

You can watch the log files in `$DIR/var/log/nl_{parser,loader,pipeline}.log` with `tail -f $DIR/var/log/nl_*.log`.

### 2.5.3 Manually loading data into the pipeline

One illustrative situation in using the pipeline is the case where you have an existing pipeline running and would like to add in data from an already collected source that you haven't been parsing and loading.

The first thing to do is to stop the existing pipeline. Assuming you are in the top-level nl\_pipeline config dir:

```
kill `./var/run/nl_pipeline.pid`
```

Then use the parser from the command line to manually parse the new source log file:

```
nl_parser -m gridftp -p one_event=True -e "(?P<ts>\S+) (?P<host>\S+) (?P<app>\S+) (: )?" / ←
    opt/osg/var/service/2009.03/gridftp.log > gridftp_parsed.log
```

In this case we are using the gridftp parser with the `one_event` parameter turned on and using a regex expression to strip off the syslog-ng header before parsing it.

Similarly from the command line, load that parsed file into the db:

```
nl_loader -u mysql://localhost -p read_default_file='./my.cnf' -p db=nldb -g < ←
    gridftp_parsed.log
```

This loads into mysql on our local host into the `nldb` database using user and password info from `./my.cnf`, showing progress as it goes.

If you want to continue to parse this file, add this new file and parser into your `nl_parser.conf` (details on how to do this are in the manual and example pipeline config files can be found under the `python/examples/nl_pipeline/` dir in the distribution).

Now start up the pipeline again:

```
nl_pipeline -c ./etc
```

## 2.6 Miscellaneous tasks

### 2.6.1 Dumping and reloading a MySQL database

Dump the database

```
$ mysqldump fake > fake.dmp
# Alternatively, to name the database in the dump file
$ mysqldump -B fake > fake.dmp
```

To install the dump, do the following

```
$ mysql -e "create database fake"
$ mysql -D fake < fake.dmp
# Or, if -B was used during dump:
$ mysql < fake.dmp
```

### 2.6.2 Using Syslog-**ng**

tbd

## 3 NetLogger Analysis

This section provides some examples of NetLogger analysis.

### 3.1 Lifelines

One can create NetLogger “lifelines” with the `nl_lifeline` program (in the Python part of the NetLogger distribution). This program takes a log file on standard input and produces a second program, with data embedded, that can be run by either `gnuplot` or `R` to produce the plot.

In the example below, the event sequence is `a`, `b`, `c`, and the “item” field is used to group events into lines, i.e. a sequence of events having the same value for `item` will be placed in the same line.

```
$ cat file.log
ts=2009-04-17T19:52:17.558026Z event=a level=Info item=1
ts=2009-04-17T19:52:21.790675Z event=b level=Info item=1
ts=2009-04-17T19:52:26.566290Z event=c level=Info item=1
ts=2009-04-17T19:52:30.992154Z event=a level=Info item=2
ts=2009-04-17T19:52:35.942049Z event=b level=Info item=2
ts=2009-04-17T19:52:47.864045Z event=c level=Info item=2

# Create plot using R
$ nl_lifeline -l item -g item -e a,b,c -o file.rplot -t R < file.log
# (output)
To create the final plot, run:
R CMD BATCH file.rplot
# Note: output will be in plot.pdf

# Create plot using Gnuplot
$ nl_lifeline -l item -g item -e a,b,c -o file.gnuplot -t g < file.log
# (output)
To create the final plot, run:
gnuplot file.gnuplot
# Note: output will be in plot.png
```

## 3.2 Database queries

This section provides some sample database queries.

### 3.2.1 Database event types

This query reports database event types and their attributes: Some of the syntax is MySQL-specific, in particular the *group\_concat* function.

```
--  
-- Put event types in a temporary table  
  
create temporary table event_types (id integer, name varchar(255))  
select id,  
       count(*) 'num',  
       min(time) 'first',  
       max(time) 'last',  
       (case  
           when startend = 0 then concat(name,'.start')  
           when startend = 1 then concat(name,'.end')  
           else name  
       end) name  
  from event group by name, startend;  
  
--  
-- Join with attr(ibutes) table  
  
create temporary table event_attrs (event varchar(255), names varchar(4096))  
select e.name as 'event',  
       group_concat(attr.name order by attr.e_id separator ',') as 'names'  
from event_types e  
  left join attr on e.id = attr.e_id  
group by e.name;  
  
--  
-- Join with ident(ifiers) table  
  
create temporary table event_ids (event varchar(255), names varchar(4096))  
select e.name as 'event',  
       group_concat(ident.name order by ident.e_id separator ',') 'names'  
from event_types e  
  left join ident on e.id = ident.e_id  
group by e.name;  
  
--  
-- Join with DN table  
  
create temporary table event_dn (event varchar(255), has_dn varchar(3))  
select e.name as 'event',  
       (case when isnull(dn.id) then 'no' else 'yes' end) 'has_dn'  
from event_types e  
  left join dn on e.id = dn.e_id  
group by e.name;  
  
--  
-- Join with text table  
  
create temporary table event_text (event varchar(255), has_text varchar(3))  
select e.name as 'event',  
       (case when isnull(text.id) then 'no' else 'yes' end) 'has_text'  
from event_types e  
  left join text on e.id = text.e_id
```

```
group by e.name;

--
-- Project them all into the same table
--
select x.event, e.num, from_unixtime(e.first, "%Y-%m-%dT%H:%i:%S") 'first', from_unixtime(e ←
    .last, "%Y-%m-%dT%H:%i:%S") 'last', x.names 'attributes', y.names 'identifiers', d. ←
    has_dn, t.has_text
from eventAttrs x
join event_ids y on x.event = y.event
join event_types e on x.event=e.name
join event_dn d on d.event=e.name
join event_text t on t.event=e.name
order by num;
```

## 4 Logging API examples

This section contains code samples for the NetLogger API, organized by language.

### 4.1 C API

#### 4.1.1 Basic usage

```
/*
Copyright (c) 2004, The Regents of the University of California, through
Lawrence Berkeley National Laboratory (subject to receipt of any required
approvals from the U.S. Dept. of Energy). All rights reserved.
*/
/** @file
 * @ingroup examples */

/** 
 * Sample high-level C API usage. This is the API that most people
 * would be expected to use.
 */
#include <stdio.h>
#include "nl.h"

/** 
 * Pretend to do something. Actually a no-op.
 */
void do_something(void)
{
    return;
}

/** 
 * MAIN
 */
int main(int argc, char **argv)
{
    NL_log_T environment, screen, debug;
    int i;

    /* If the destination is NULL, setting
     * the environment variable 'NL_DEST' will redirect it, e.g.:
```

```
*      % export NL_DEST=/tmp/myData.log
*/
environment = NL_open(NULL);

/* Set destination to stderr, output level to 'info', and
 * add a string constant to every message
 */
screen = NL_open("&");
NL_set_level(screen, NL_LVL_INFO);
NL_set_const(screen, "constant:s", "constant value");

/* Debugging handle */
debug = NL_open("&");
NL_set_level(debug, NL_LVL_DEBUG1);

/* Write an empty 'program.start' event to the 'screen' log. */
NL_write(screen, NL_LVL_INFO, "program.start", "");

/*
 * log a 'loop.start' event with two values to the 'debug' log.
 * since we know the values are constant, we can use a ':'
 * instead of an '=' as the keyword:datatype separator.
 *
 * By default, these will not be logged, but could be
 * "turned on" by using the dynamic levels feature.
 * e.g., if you execute the (bash) shell commands
 *      % export NL_CFG=/tmp/levels.cfg
 *      % echo 5 > /tmp/levels.cfg
 * before running this program, you will see the messages.
 */
NL_write(debug, NL_LVL_DEBUG, "loop.start", "start:i stop:i", 0, 10);
for (i = 0; i < 10; i++) {
    /*
     * Wrap do_something() with an eponymous event at
     * a higher (more fine-grained) debugging level.
     *
     * To turn on these messages, you will need to set the
     * logging level to '6' (see comment above).
     */
    NL_write(debug, NL_LVL_DEBUG1, "do.something.start", "index=i", i);
    do_something();
    NL_write(debug, NL_LVL_DEBUG1, "do.something.end", "index=i", i);
}
/*
 * these are just mirror images of the loop.start/program.start
 * messages
 */
NL_write(debug, NL_LVL_DEBUG, "loop.end", "start=i stop=i", 0, 10);
NL_write(screen, NL_LVL_INFO, "program.end", "", 1.0);

/* close logs */
NL_close(environment);
NL_close(debug);
NL_close(screen);

return 0;
}
```

#### 4.1.2 Format a string

Shows how to create formatted NetLogger strings.

```
#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>
#include "nl.h"

int main(int argc, char **argv)
{
    int len;
    char buf[65536];
    struct timeval start_t, end_t;
    gettimeofday(&start_t, 0);
    gettimeofday(&end_t, 0);
    len = NL_sprintf(buf, "gridftp.FTP_INFO", NL_LVL_INFO,
                      "HOST=s PROG=s "
                      "START.TM=d END.TM=d "
                      "USER=s FILE=s "
                      "BUFFER=l BLOCK=l NBYTES=l VOLUME=d "
                      "STREAMS=i STRIPES=i "
                      "DEST=s TYPE=s CODE=i",
                      "131.243.2.1", "GridFTP",
                      start_t.tv_sec + start_t.tv_usec / 1e6,
                      end_t.tv_sec + end_t.tv_usec / 1e6,
                      "Elmo", "SesameStreet.txt",
                      1LL, 2LL, 32767LL, 1e9,
                      8, 12, "123.456.3.2", "what", 1);
    write(STDOUT_FILENO, buf, len);
}
```

#### 4.1.3 Transfer API

How to use the NetLogger *transfer* API, designed for easily and efficiently logging details of data transfers.

```
/*
Copyright (c) 2007, The Regents of the University of California, through
Lawrence Berkeley National Laboratory (subject to receipt of any required
approvals from the U.S. Dept. of Energy). All rights reserved.
*/
static const volatile char rcsid[] = "$Id$";
/** @file nltransfer_example.c
 *  @ingroup nlsumm examples
 */
/*@{ */

/**
 * Example usage of the NetLogger "transfer" API.
 * See usage() function for program arguments.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "nltransfer.h"

/** In reality, blocks would probably be O(100K) */
```

```
#define BLOCK_SIZE 8

/* Pretend to read some data */
int read_bytes(void)
{
    usleep(1);
    return BLOCK_SIZE;
}

/* Pretend to write some data */
int write_bytes(void)
{
    usleep(20);
    return BLOCK_SIZE;
}

/* With apologies to Mark Harris */
int bang_the_drum_slowly(void)
{
    usleep(500000);
    return BLOCK_SIZE;
}

/* Prototype for run function */
typedef int (*run_fn_t)(const char *, int);

/*
 * Get GUID from NetLogger
 */
void get_my_guid(char **guid)
{
    if (!NL_get_guid(guid) && !NL_get_someid(guid)) {
        *guid = strdup("00000000-0000-0000-0000-000000000000");
    }
}

/**
 * Log everything to the same place.
 */
int run1(const char *logfile, long summ_usec )
{

    char err_why[1024]; /* for error reporting */
    char *guid;
    NL_summ_T summ;
    NL_log_T log;
    const char *stdout_logfile = "-";
    char *result_str;

    /* Get a GUID */
    get_my_guid(&guid);
    /* Set log file to stdout if NULL */
    if (!logfile)
        logfile = stdout_logfile;

    /*
     * (1) Create NetLogger log instance
     */
    /* Open log, return on error */
    log = NL_open(logfile);
    if (!log) {
        sprintf(err_why, "opening '%s'", logfile);
```

```
        goto error;
    }
/* Set log to DEBUG level */
NL_set_level(log, NL_LVL_DEBUG);

/*
 * (2) Create summarizer
 */
summ = NL_summ();
/* Make summarizer output go to log, too */
NL_summ_set_shared_output(summ, log);
/* Add NL_transfer events to summarizer.
   Set interval to -1 to mean "until NL_finalize()".
 */
NL_transfer_init(summ, summ_usec, NL_LVL_DEBUG);
/* Don't let summarizer 'consume' the events,
 * we want them to "pass through" and get logged
 * in their full detail.
 */
NL_transfer_set_passthrough(summ);
/* Add this summarizer to log instance */
NL_summ_add_log(summ, log);

/*
 * (3) Log some activity
 */
NL_write(log, NL_LVL_INFO, "nltransfer_example.activity.start", "");
{
    NL_transfer_blockid_t block_id;
    NL_transfer_streamid_t stream_id;
    NL_transfer_op_t op;
    double nbytes;

    for (block_id = 0; block_id < 2; block_id++) {
        for (stream_id = 11; stream_id < 13; stream_id++) {
            int opctr;
            for (opctr=NL_TRANSFER_DISK_READ; opctr < NL_TRANSFER_NOEVENT; opctr++) {
                op = (NL_transfer_op_t)opctr;
                NL_transfer_start(log, NL_LVL_DEBUG, op,
                                  guid, stream_id, block_id);
                nbytes = bang_the_drum_slowly();
                NL_transfer_end(log, NL_LVL_DEBUG, op,
                               guid, stream_id, block_id, nbytes);
            }
        }
    }
}
NL_write(log, NL_LVL_INFO, "nltransfer_example.activity.end", "");

/*
 * (4) Finalize handles and get result string.
 * Then close everything up
 */
NL_transfer_finalize(summ);
result_str = NL_transfer_get_result_string(summ, guid);
printf("RESULT STRING:\n%s", result_str);
NL_summ_del(summ);
NL_close(log);

return 0;

error:
```

```
fprintf(stderr, "Error:%s:%s\n", err_why, NL_err_str());
return -1;
}

/***
 * Print usage message and exit
 */
void usage(void)
{
    printf("Program usage: nltransfer_example [-o logfile] [-i usec]\n");
    exit(0);
}

/***
 * Program entry point
 */
int main(int argc, char **argv)
{
    long usec;
    int i, result = 0;
    char *logfile, *endptr;

    logfile = NULL;
    usec = -1;
    if (argc >= 2) {
        for (i=1; i < argc-1; i++) {
            if (argv[i][0] != '-')
                usage();
            switch (argv[i][1]) {
                case 'i':
                    usec = strtol(argv[i+1], &endptr, 10);
                    if (endptr == NULL)
                        usage();
                    break;
                case 'o':
                    logfile = strdup(argv[i+1]);
                    break;
                default:
                    usage();
            }
        }
    }

    if (usec < 0)
        usec = 1000000;
    if (!logfile)
        logfile = strdup("-");

    printf("-----\n");
    result = run1(logfile, usec);

    free(logfile); /* keep valgrind happy */

    return result;
}

/*@ */
```

## 4.2 Java API

### 4.2.1 Basic usage

Basic Java API example.

```
import gov.lbl.netlogger.LogMessage;
import java.util.UUID;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Example use of NetLogger.
 */
public class NetLoggerExample {

    private String guid;
    private Log log;
    private static int COUNT = 2;

    /**
     * Constructor
     */
    public NetLoggerExample() {
        this.guid = UUID.randomUUID().toString();
        this.log = LogFactory.getLog(NetLoggerExample.class);
    }

    /**
     * Make a short example log without log4j.
     */
    public void nolog4j() {
        LogMessage m;
        for (int i = 0; i < COUNT; i++) {
            m = new LogMessage("nolog4j.start").add("guid",guid).add("count",i);
            print(m);
            m = new LogMessage("nolog4j.end").add("guid",guid).add("count",i);
            print(m);
        }
    }

    /**
     * Make a short example log with log4j.
     */
    public void withlog4j() {
        LogMessage m;
        for (int i = 0; i < COUNT; i++) {
            m = new LogMessage("log4j.start").add("guid",guid).add("count",i);
            this.log.info(m);
            m = new LogMessage("log4j.end").add("guid",guid).add("count",i);
            this.log.info(m);
        }
    }

    /**
     * Program entry point.
     *
     * @param argv
     */
    public static void main(String[] argv) {
        NetLoggerExample ex = new NetLoggerExample();
        ex.nolog4j();
    }
}
```

```

        ex.withlog4j();
    }

    /**
     * Helper function to print to console.
     */
    private static void print(LogMessage m) {
        System.out.println(m.toString());
    }

}

```

#### 4.2.2 Log4J

##### Sample Log4J configuration with for NetLogger

```

#####
# log4j logging configuration file.
# to enable log4j, run java with
#      java -Dlog4j.configuration=file:path/to/log4j.properties ...
#####

log4j.rootLogger=INFO

# Send 'NetLoggerExample' messages to the NETLOGGER appender
log4j.logger.NetLoggerExample=DEBUG, NETLOGGER

# - Append to file
#log4j.appenders.NETLOGGER=org.apache.log4j.FileAppender
#log4j.appenders.NETLOGGER.File=netlogger-example.log
#log4j.appenders.NETLOGGER.bufferedIO=true
#log4j.appenders.NETLOGGER.bufferSize=16384

# - Append to console
log4j.appenders.NETLOGGER=org.apache.log4j.ConsoleAppender

# Layout for NETLOGGER appender
log4j.appenders.NETLOGGER.layout=org.apache.log4j.PatternLayout
log4j.appenders.NETLOGGER.layout.ConversionPattern=%m\n

# Layout for Console appender
log4j.appenders.Console=org.apache.log4j.ConsoleAppender
log4j.appenders.Console.layout=org.apache.log4j.PatternLayout

```

### 4.3 Perl API

#### 4.3.1 Basic usage

##### Basic Perl API example

```

use NetLogger;
use Carp;

$h = {
    val => "1",
};

$log = NetLogger->new();
$log->set_level($NetLogger::INFO);

```

```
# Write messages to 'foo.log'
$log->open("foo.log");
$log->debug("hello", $h);
$log->warning("goodbye", $h);
# Write the same messages to a socket
$r = $log->open("x-netlog://www.mit.edu:123");
print "log open silently failed\n" unless $r;
$log->debug("hello", $h);
$log->warning("goodbye", $h);
# Write the to stdout as well
$log->open("-");
$log->info("info.message", $h);
# Now try UDP
$r = $log->open("x-netlog-udp://localhost:15000");
print "UDP log open silently failed\n" unless $r;
$log->debug("hello", $h);
$log->warning("goodbye", $h);
```

## 4.4 Python API

### 4.4.1 Basic usage

Python "logging" API

```
"""
Logging Best Practices messages in Python.

"""

import logging
from netlogger import nllog

# Instead of logging.getLogger(), use nllog.getLogger()
log = nllog.getLogger("my.program")
# Otherwise proceed as usual
log.addHandler(logging.StreamHandler())
log.setLevel(logging.INFO)
# Log a message with event name as first argument,
# then keyword=value pairs for the rest. Use
# double underscore to mean '.'
log.info("hello", to__whom="world", from__whom="me")
```

---

#### Example 4.1 Output from logging\_example.py

```
ts=2008-09-24T22:23:30.402704Z event=my.program.hello level=INFO from.whom=me to.whom=world
```

---

### 4.4.2 Format strings

Using the standalone module *bpize.py* to format messages in a way that is easily integrated with the Python standard logging library.

```
"""
Example of bpize.py module
"""

import logging
from netlogger import bpize

# set up logger
```

```
logging.basicConfig()
for hndlr in logging.getLogger().handlers:
    bpize.BPize(hndlr, namespace="test.")
log = logging.getLogger("bpize_example")
log.setLevel(logging.INFO)
# write message
log.info('did.foo bar=purple baz="red, white and blue"')
```

---

**Example 4.2** Output from bpize\_example.py

```
ts=2008-09-24T15:15:37.044-08:00 guid=506b4f0a-8a86-11dd-8e9a-001b63926e0d event=test.did. ↵
foo bar=purple baz="red, white and blue" level=INFO
```

---